HAUST 2016

ÞÝÐENDUR

T-603-THYD

Homework 3

Nemandi:

Steinn Elliði Pétursson

Kennitala:

250594-2759

26. október 2016

Kennari:

Friðjón Guðjohnsen

1

Consider the following grammar:
$E \rightarrow E\ T\ O$
$E \rightarrow T$
$O \rightarrow +$
$O \rightarrow -$
$T \rightarrow num$
where "+". "-" and num are tokens. For simplicity all numbers are single digit.

a)

Write a syntax-directed definition (SDD) for the grammar that changes a postfix expression to prefix form. Assume that each nonterminal has the attribute val of type string, and that each attribute value of a node in the parse tree denotes a sub-expression in prefix form.

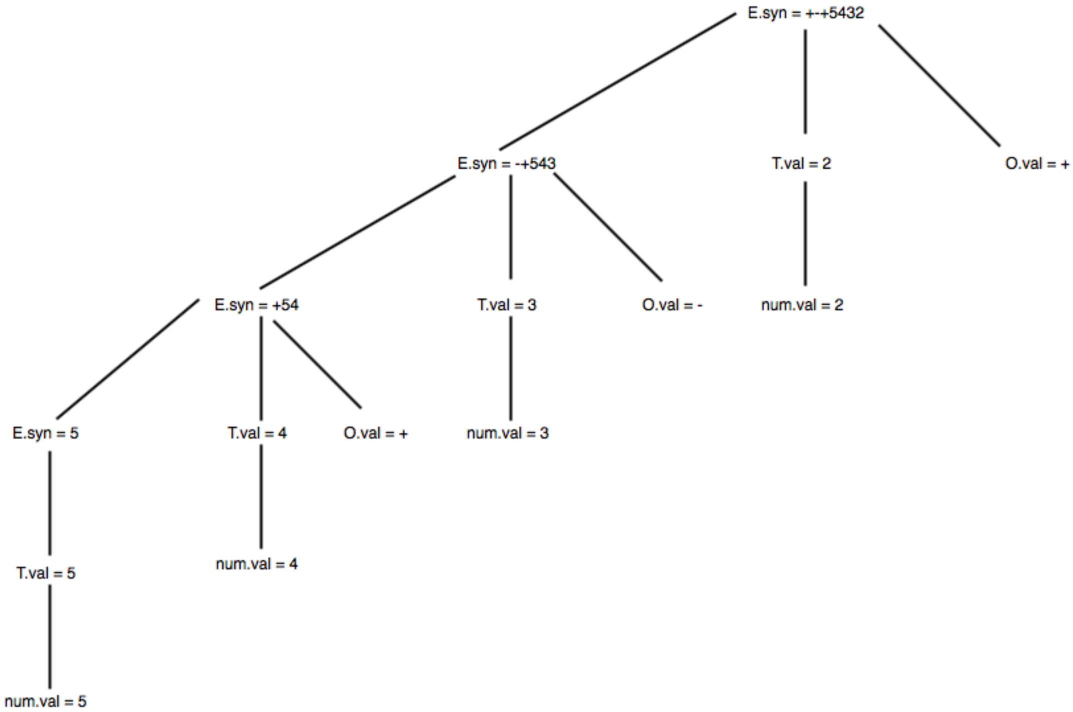
Solution.

Production	Semantic rules
$E \rightarrow E_1\ T\ O$	$E.val = O.val\ E_1.val\ T.val$
$E \rightarrow T$	$E.val = T.val$
$O \rightarrow +$	$O.val = +$
$O \rightarrow -$	$O.val = -$
$T \rightarrow$ **num**	$T.val =$ **num**.val

b)

Annotate a parse tree for input string 54+3-2+. Note that the attribute value of the root of the parse tree should show the input string in prefix form for the whole expression.

Solution.



2

Consider the following grammar for type declarations:

$D \rightarrow id\ L$
$L \rightarrow id\ L \mid\ : T$
$T \rightarrow integer \mid real$

a)

For this grammar, write an SDT (Syntax-Directed Translation Scheme) which sets the type for each name into the symbol table. Use the function addType(X,Y), for which X is a reference/pointer to a symbol table entry and Y is a type.

Production	Actions
$D \rightarrow id\ L$	$L\ \{addType(id, L.type)\}$
$L \rightarrow id\ L_1$	$L_1\ \{addType(id, L_1.type)\ L.type = L_1.type\}$
$L \rightarrow : T$	$T\ \{L.type = T.val\}$
$T \rightarrow integer$	$\{T.val = integer\}$
$T \rightarrow real$	$\{T.val = real\}$

b)

Write a recursive-descent parser for the translation scheme you developed. You can assume the function yylex(), which returns the next token from the lexical analyzer. Moreover, assume the function match(Token t), which checks whether the current token matches token t and calls yylex() if that is the case, otherwise it reports an error. In the solution, you should let some functions (that correspond to some non-terminals) return a type. Assume that the type is an integer constant in the form of an enumeration:
$enum\ TypeCode\ \{INTEGER, REAL, ERROR\}$

Solution.

```
void D() {
        match(tc_id);
        addToken(currentToken, L());
}

int L() {
        if (match(tc_id)) {
                string token = currentToken;
                int type = L();
                addType(token, type);
                return type;
        } else if (match(tc_colon)) {
                return T();
        } else {
                handleError("expected id or colon");
                return 2;
        }
}

int T() {
        if (match(tc_integer)) {
                return 0;
        } else if (match(tc_real)) {
```

```
                return 1;
        } else {
                handleError("expected type");
                return 2;
        }
}
```

3

Construct a sequence of TAC instructions (op, arg1, arg2, result) for the statement:

$z = (a + b) * ((c + d) - (-a + b + c))$

Note that * has higher precedence than +. Construct the code in the same way as your own top-down parser would do, i.e. use your changed grammar (the grammar for the Decaf language) to obtain the right precedence order. Moreover, make sure that the order of quadruples mirrors the order that will be generated by your parser.

Solution.

VAR	t1		
ADD	a	b	t1
VAR	t2		
ADD	c	d	t2
VAR	t3		
UMINUS	a	t3	
VAR	t4		
ADD	t3	b	t4
VAR	t5		
ADD	t4	c	t5
VAR	t6		
SUB	t2	t5	t6
VAR	t7		
MULT	t1	t6	t7
APARAM	t7		
CALL	writeln		
RETURN			

4

Construct a sequence of TAC instructions (op, arg1, arg2, result) for the code fragment:

```
{
        if (i < j) {
                j = j * 2;
        } else {
                j = j / 2;
        }
}
```

Solution.

	LT	i	j	if
	VAR	t1		
	DIV	j	2	t1
	ASSIGN	t1	j	
	GOTO	ret		
if:	VAR	t2		
	MULT	j	2	t2
	ASSIGN	t2	j	
ret:	APARAM	j		
	CALL	writeln		

5

Construct a sequence of TAC instructions (op, arg1, arg2, result) for the code fragment:

```
{
int n;
    int sum;
    sum = 0;
    for(n=0;n<10;n++) {
       sum = sum + n * n;
    }
}
```

Solution.

	VAR	n		
	VAR	sum		
	ASSIGN	0	sum	
	ASSIGN	0	n	
for:	GE	n	10	ret
	VAR	t1		
	MULT	n	n	t1
	VAR	t2		
	ADD	sum	t1	t2
	ASSIGN	t2	sum	
	VAR	t3		
	ADD	1	n	t3
	ASSIGN	t3	n	
	GOTO	for		
ret:	APARAM	sum		
	CALL	writeln		

6

What do MSIL (Microsoft Intermediate Language) and Java byte- code have in common? What are their differences? Use whatever sources you want to find answers to these questions, but make sure that you provide references to your sources in proper scholarly fashion.

Solution.

Both MSIL and the Java byte-code are CPU independent intermediate languages compiled from higher level programming languages defined in a stack based virtual environments. While MSIL is always compiled (with Just in time compilation) the Java byte-code can both be compiled and interpreted. Java byte-code consists solely of 1 byte opcodes except for 2 opcodes dealing with table jumping, MSIL also consists solely of 1 and 2 byte opcodes.

References:

`http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-1.html#jvms-1.1`
`http://aspalliance.com/1123_Understanding_the_Microsoft_Intermediate_Language.8`
`http://www.javaworld.com/article/2077233/core-java/bytecode-basics.html`